Amiga Mail News

The Amiga 600

At the CeBit computer fair held last March in Germany, Commodore previewed the Amiga 600, the new consumer-level Amiga model. It takes advantage of several technologies not previously seen in the Amiga line.

The A600 is much smaller than the A500 thanks to Surface Mount Technology and smaller casework. Surface Mount Technology (SMT) makes it possible to reduce the size of the motherboard, lowering cost while at the same time increasing the reliability of the circuitry. There is only one socketed IC on the A600, the Kickstart ROM.

The A600 has a smaller keyboard than its cousin the A500. The A600 keyboard contains all the QWERTY-style keyboard keys and the function keys. The major difference is that the A600 keyboard lacks a numeric keypad.

This new Amiga contains either one or two surface-mounted, 256Kx16 DRAMs which give the A600 either 512K or 1 Megabyte of RAM. The A600 can handle up to a total of 2MB of RAM internally with the help of an internal add-on called the A601. The A601 also contains a battery backed-up real time clock.

The A600 contains a surface mounted ECS chip set. The A600 can take advantage of all of the display modes available in higher end Amiga systems like the A3000. This includes the 31 KHz Productivity mode, although this mode requires an appropriate monitor. Also, with an A601 installed, the A600 can address up to two Megabytes of Chip RAM.

Because the A600 is a consumer-oriented product, it contains some connectors to allow the user to connect it to a TV or video equipment. The Amiga 600 has an RF modulator built into it. It also has color composite video out. Both of these connectors are in the form of standard RCA jacks.

One major addition the A600 introduces to the low-end Amiga is a built-in hard drive interface. The A600 has a connector for an internal Integrated Drive Electronics (IDE) hard drive. IDE hard drives, which are commonly found in laptop PCs, use very little power and are only 2.5" tall. There is enough space inside the A600 to house one of these drives. Installing an IDE drive requires either a ROM upgrade kit or a software loaded IDE device driver.

The A600 comes with a special version of the

IN THIS ISSUE

Debugging with Enforcer and Mungwall
Carolyn Scheppner

SANA-II Network Device Specification
Brian Jackson, Dale Larson, Greg Miller, and Kenneth Dyke

Four Can Play–Supporting Parallel Port Joysticks
Christian Ludwig

Making CDTV Titles That Work on Amigas
Ben Phister

operating system software, 2.05. The 2.05 ROM for the A600 contains drivers for the A600's PCMCIA credit card connector. This new connector is compatible with the PCMCIA/JEIDA standard for "Memory Card" devices. The PCMCIA connector supports credit-card sized memory and I/O cards.

This 68 pin PCMCIA 2.0 connector has many possible uses. Game programmers and other application programmers can design applications which run, in place, from PCMCIA standard ROM cards. Using an appropriate memory card, the user can add up to four Megabytes of RAM to the system. Using this same type of card and a special version of CrossDOS, the user can share data with other machines that support PCMCIA, like the HP95XL Palmtop computer.

The PCMCIA spec is available to registered developers through CATS US. Check your developer support material price lists for this item. There are also some industry magazines which deal exclusively with memory card devices. One such publication is "Memory Card Systems and Design", 6300 S. Syracuse, Suite 650, Englewood, CO, 80111-9912. CATS will also be providing some tools and examples for Amiga XIP (execute-in-place) credit card software.

If you have a software product on the Amiga market that is not Release 2.0 compatible, get an upgrade out there that is compatible. As the entire installed base of Amigas shifts to Release 2.0 (both new Amiga models and old), any product that isn't Release 2.0 compatible essentially will be rendered worthless. Any developers that are wondering when they start developing specifically for Release 2.0, the answer is *right now*.

At this time Commodore is not marketing the A600 in the United States.

Hirsch and Wolf are in Business

European Amiga developers can now buy developer materials directly from Germany. CATS Europe has established a European-wide service via Hirsch & Wolf in Germany, where developers can order articles from CATS official material list.

Hirsch & Wolf oHG
Attn: Hans-Helmut Hirsch
Mittelstr. 33
5450 Neuwied 1
Germany

Fax: ++49-2631-23878
Tel: ++49-2631-24485
UUCP: cbmger!carla!hhhirsch

Hirsch and Wolf accept several forms of payment including international money order, EuroCard, Master Card, and Visa. For more information contact Hirsch and Wolf or your local support manager.

When is the Next DevCon?

CATS is making plans for the next Amiga Developers Conference. Although the plans aren't finalized yet, we did want to let you know that CATS is planning the next conference for January, 1993. There are several reasons for picking a January date, which we will make clear once all the arrangements have been made.

So, be assured that CATS is working on the next conference, and it will be in a location as interesting (or more so) than our previous DevCons!

Q: The Autodoc for the Intuition function `ActivateWindow()` *says:*

```
* RESULT
*    V35 and before: None.
*    V36 and later: returns zero if
*    no problem queuing up
*    the request for deferred action
```

Is this true?

A: No, it's actually none, even under V36, V37, etc.

Q: If I use the trackdisk.device to write on a section of a write-protected floppy, when I read that section of the floppy, the data I wrote appears to be there. What is going on?

A: When writing to a write-protected floppy using the *trackdisk.device*, the *trackdisk.device* does not write on the disk, but it does write to the disk buffer in memory, which is what you are reading. This is a bug.

In order to make sure that the state of the disk is as you expect after a failed write, you should do a `CMD_CLEAR` to make it flush the buffer.

This is not normally a problem with the file system, since it checks write-protect on every insertion, and doesn't attempt writes to write-protected disks.

Q: What's wrong with calling the Exec function `AllocMem()` *using the* `MEMF_REVERSE` *flag?*

A: Under normal conditions, the `MEMF_REVERSE` flag makes `AllocMem()` search Exec's free memory list in reverse order. If the `MEMF_REVERSE` allocation fails due to low memory, the OS will either clear low memory or get stuck in an infinite loop (or, when *Enforcer* is running, it will cause a number of Enforcer hits!)

Workaround:

If you really want to do this and don't want to have to do the `MEMF_REVERSE` yourself, you can do the following workaround. It is not very fast but if your allocations are rare, it will not be too bad.

```
Forbid();
if (mem=AllocMem(size,
    <normal flags, no MEMF_REVERSE> ))
{
     flags=TypeOfMem(mem);
     FreeMem(mem,size);
     mem=AllocMem(size,
                  MEMF_REVERSE|flags);
}
Permit();

if (mem)
{
     /* Got the memory... */
}
else  /* Failed! */
```

Warning: This will only work if there is only one memory list with the attributes given (which is usually the case with `MEMF_CHIP`). If there are more than one memory lists, `AllocMem()` may

work in the second list while the reverse will fail in the first (and crash).

Warning: Tools such as *Memoration* can cause errors in the second `AllocMem()` from the workaround above.

This bug exists in all versions of Exec to date.

Q: The Autodoc for the DOS function `InternalLoadSeg()` *states that* `ReadFunc()` *takes it arguments in registers d1/a0/d0. Is that true?*

A: No, it actually takes them in registers d1/d2/d3.

Q: Does the input.device ever try to lock the blitter?

A: Sure, all the time. All input handlers run on the *input.device* task, and the grandest input handler of all is called "Intuition". When an application calls Intuition, part or most of the function executes on the application's task, but part may execute on the *input.device* task. All user-initiated actions (e.g., dragging a window) always happen on the *input.device* task. This means the *input.device* does rendering, layer operation, copper-list and ViewPort operations, etc.

Q: I program in assembler. I hear that many software compatibility problems are traced to assembler application code containing a hidden misuse of a register. How can I check for this?

A: While programming in assembler, it is not uncommon for programmers to forget to refresh a scratch register (d1/a0/a1) after a system call, or even look at the wrong register for the result of the system call. These registers contain leftover values from the internal code of the system function, which may *happen to be* the original value which was in the register before the call, or *may happen to be* a copy of the result (d0). If this is the case, the assembler application's register misuse bug may have no symptoms or only sporadic symptoms under one version of the OS. However, the slightest change to the system function's internal code can drastically change the leftover values in the scratch registers. In some cases, *one* instance of register misuse can render a major application *unusable* under a new version of the OS.

Here is a simple example of such a hidden coding error:

```
* GfxBase already in A6.  Both SetDrMd and
* SetAPen expect a rastport pointer in A1
    MOVEA.L rastport, a1    * Put rport in A1
    MOVE.L  #JAM1, d0       * JAM1
    JSR _LVOSetDrMd(a6)     * set draw mode
    MOVE.L  #3, d0          * pen 3
* Here's the problem: the programmer assumes
* A1 still contains the rastport pointer.
* Since A1 is a scratch register, SetDrMd
* may have overwritten A1 with garbage, so
* SetAPen will get a bogus RastPort pointer.
    JSR _LVOSetAPen(a6)     * set pen
```

If the rastport pointer passed in A1 *happens to be* left over in A1 after the call to `SetDrMd()`, the call to `SetAPen()` will succeed. If not, the call to `SetAPen()` will trash memory, and possibly crash the system.

If you program is assembler, you *must* test your code with *Scratch* (by Bill Hawes) to test for misuse of registers after system calls. *Scratch* and the script that installs it (*scratchall.script*) are on the Software Toolkit II disk of the *2.0 Native Developer Update*. It may also be found with the debugging tools on the Denver/Milan Devcon disks. *Scratch* will *invalidate* the scratch registers upon the exit from each system library call. If a program is failing to refresh a scratch register or looking at a scratch register improperly, you may get Enforcer hits (if you are running *Enforcer and Scratch*), and/or Mungwall hits, and/or obvious misbehavior or crashing of your code.

Use the *scratchall.script* to install *Scratch* before starting *Mungwall*. When running running this script watch out for the scripts with a backtick. Some versions of the script have a backtick (`) at the beginning of a early comment line. The script will not execute unless the backtick is replaced with a semicolon (;).

Amiga Mail
Volume II
May/June 1992

Debugging with Enforcer and Mungwall

by Carolyn Scheppner

It is almost impossible for a human being to develop software without including some bugs. Some bugs make themselves known rather quickly. Other bugs are not so easy to notice. For example, two hard to find bugs that sometimes slip past developers are using uninitialized pointers and using memory which has already been freed. These bugs often reference semi-random memory, which contains semi-random data. Typically the behavior of the software depends on the data passed to it. If that data is erratic, the behavior of the software will also be erratic. Because the behavior is erratic and unpredictable, the problem is very hard to spot. Quite often, bugs like these go unnoticed until software is running in a real user's multitasking environment. Fortunately, debugging tools like *Enforcer* and *Mungwall* (and for assembler programmers, *Scratch* by Bill Hawes) help uncover hidden bugs.

Enforcer is a debugging tool written by Bryce Nesbitt. Its job is to report any attempts to access regions of the Amiga address space that are off-limits to applications. These off-limits accesses include reading and writing to the lowest 1K range of memory (except for reading ExecBase from $4), writing to the Kickstart ROM, and reading and writing to non-existent memory ranges.

Enforcer requires that the CPU has access to an MMU (Memory Management Unit) to catch reads and writes to memory. Most 68020 based accelerator boards have MMU chips on them. An MMU is built-in to each 68030 and 68040 CPU (currently *Enforcer* does not support 68040).

When an application tries to read a memory location that has been read-protected by *Enforcer*, *Enforcer* intercepts that memory read, reports the illegal memory access (known as an "Enforcer hit"), and shows the application a zero instead of the contents of the memory address. The application has no idea that *Enforcer* did anything. When an application tries to write to a memory location that has been write-protected by *Enforcer*, *Enforcer* prevents the illegal memory write and issues an Enforcer hit.

Enforcer is even more powerful when used in combination with *Mungwall*. *Mungwall* was written by Ewout Walraven and is based on *Memmung* by Bryce Nesbitt and *Memwall* by Randell Jesup. The "mung" part of *Mungwall* fills all of free memory (and all subsequently freed memory) with a large, odd, 32-bit value. An odd value is likely to cause serious problems for any program that uses wild or uninitialized pointers, or uses memory after it has been freed.

Unlike *Enforcer*, *Mungwall* does not require any special hardware. *Mungwall* can run without *Enforcer* and on non-MMU machines.

Mungwall uses several special 32-bit values to "mung" memory which helps diagnose problems:

Except when *Enforcer* is running, *Mungwall* sets location zero to $C0DEDBAD. Normally, location zero is $00000000. By putting an odd, non-zero value in location zero, any erroneous references to location zero are much more likely to show themselves. For example, a program that references location zero as character array will see a string that starts with the ASCII values $C0 $DE $DB $AD, rather than seeing a NULL string.

When *Mungwall* starts up, it sets all free memory to $ABADCAFE. If this number shows up while an application is running, it is likely that someone is referencing memory in the free list.

When a program allocates memory, *Mungwall* sets that memory to $DEADF00D (Except when allocating memory with MEMF_CLEAR). When an application accidentally accesses its memory before initializing it, the application will find the well-known value $DEADF00D rather than some random value that happened to be left in memory.

Mungwall fills deallocated memory with $DEADBEEF, which makes using freed memory bugs much more obvious.

The "wall" part of *Mungwall* allocates extra memory before and after every memory allocation and fills this memory "wall" with a fill pattern (normally $BB) and some information *Mungwall* uses to perform certain tests on an application's memory blocks:

When an application deallocates memory, *Mungwall* reports when the size of the deallocated memory block does not match its size when it was allocated.

Mungwall reports when a memory block's "walls" have been overwritten.

Mungwall reports allocations and deallocations of memory blocks that are zero bytes long and deallocations of memory blocks that start at location zero.

Mungwall has an option to "snoop" and report on all memory allocations and deallocations for all tasks or specific tasks. This feature can be useful when tracking down memory losses. Because the snoop option can generate so many reports, the output can be run through the snoopstrip program which will throw away all matching allocate/deallocate pairs.

The Debugging Setup

Enforcer and *Mungwall* both output their debugging information to the serial port at the baud rate to which the Amiga's serial hardware is currently set. After powerup, the serial hardware is set to 9600 baud. You can use a terminal package to alter the current baud rate setting. To set up an Amiga for serial debugging, connect your Amiga's serial port via a NULL-modem serial cable to a terminal.

The best debugging setup is to connect your Amiga to another computer running a terminal package. Ideally the terminal package should have an ASCII capture mode so it can capture all the serial debugging output and save it to a file for examination. The terminal package should also beep when it receives a CTRL-G, as both *Enforcer* and *Mungwall* send a CTRL-G beep with each Enforcer/Mungwall hit. There are several less effective alternatives to using *Enforcer* and *Mungwall* with a remote terminal. Both can send their output to a serial printer. There are special versions of both *Enforcer* and *Mungwall* that send their output to the parallel port (*Enforcer.par* and *Mungwall.par*) for output to a parallel printer. In a

pinch, attach a modem to the serial port and run a terminal package set to the modem's baud rate. As long as the modem has not made any telephone connection, the modem will bounce back any Enforcer or Mungwall hits that come through the serial port, which the terminal package can capture. Because the debugging information has to move at modem's baud rate, the modem method tends to lose data, especially when there is a lot Enforcer/Mungwall hits.

Sample Enforcer Output

Here is a sample Enforcer hit which was caused by a program called *Lawbreaker* which tried to read from location $14.

```
Program Counter (approximate)=  343F4A          Fault address       =      14
User stack pointer           =  348734          DOS process address =  339590
Data: DDDD0000 DDDD1100 DDDD2200 DDDD3300 DDDD4400 DDDD5500 DDDD6600 DDDD7700
Addr: AAAA0000 AAAA1100 AAAA2200 AAAA3300 AAAA4400 AAAA5500 AAAA6600 00002E28
Stck: 00210D70 00000FA0 00339F84 BBBBBBBB BBBBBBBB BBBBBBBB BBBBBBBB BBBBBBBB
READ-WORD  (---)(-)(-)    SR=0008   SSW=0161
Background CLI, "lawbreaker", Hunk #0, Offset $5A
```

Program Counter - Normally, this is the address of the instruction that was executing when the Enforcer hit occurred. For some types of Enforcer hits, this is the address of the instruction that executed after the hit. Note that if a program passes a bad pointer or an improperly initialized structure to a system ROM routine, it can cause the ROM code to read or write to an illegal address.

Fault Address - This is the address where the illegal access occurred. In this example, the illegal access occurred at address $14, and as specified later in the debugging output, this access was a `READ-WORD` access. So the illegal memory access was an attempt to read a word (2 bytes) at address $14. Low memory accesses are often caused by `NULL` pointers to structures. If, for example, a ROM routine references a WORD-sized structure member at offset $20, and an application passes the ROM routine a `NULL` pointer as a pointer to that structure, *Enforcer* will report that the ROM routine tried to read a word at address $20.

User Stack Pointer - This is the value that was in register A7 when the Enforcer hit occurred. It points to the top of the stack for the task that was running when the Enforcer hit occurred.

DOS Process Address - This points to the Task structure of the task that was running when the Enforcer hit occurred.

Data/Addr (Register Dump) - This is the contents of registers D0-D7 and A0-A7. This information can help assembly programmers and programmers who like to debug at a low level. Notice that register A7, the stack pointer, no longer contains the stack pointer for the task that caused the Enforcer hit (A7 does not match the user stack pointer above). Ignore the value in A7.

Stck (Stack Dump) - This is the eight long words at the top of the offending program's stack. For those who need to see more of the stack, there is a special version of *Enforcer* called *Enforcer.megastack.* that shows the last 32 long words on the stack.

Access Type - This tells what kind of memory access the Enforcer hit is. In this example the access is a `READ-WORD`, which most likely is a result of *Lawbreaker* accessing a word-sized structure member. There are two other read access types: `READ-BYTE`, which is generally a result of a bad string pointer, and `READ-LONG`, which is normally a result of a bad pointer or a bad pointer within a structure. When an errant program directly or indirectly writes over Enforcer-protected memory, *Enforcer* reports a `WRITE-BYTE`, `WRITE-WORD`, or `WRITE-LONG` type access. When *Enforcer* issues an `INSTRUCTION` type memory access, the CPU tried to load an instruction from an invalid address. Some common causes for this type of Enforcer hit are:

An errant program has overwritten a program's instructions,
An errant program has overwritten a subroutine return address that was on the stack,
The CPU tried to execute a library function using an invalid library base.

Notice that the errant program is not necessarily the program that caused the Enforcer hit.

Interrupt/Forbid/Disable - The series of parentheses after the access type indicate if the Enforcer hit occurred in an interrupt (and if so, the interrupt level), Forbid, or Disable state:

`(I-n) (-) (-)`	The hit occurred in an n level interrupt.
`(---) (F) (-)`	The hit occurred while the Amiga was in a Forbid state.
`(---) (-) (D)`	The hit occurred while the Amiga was in a Disable state.

SR (Status Register) - The CPU's status register (see 680x0 manual for more information).

SSW (Special Status Word) - The special status word for 68010 though 68040 CPUs (see 680x0 manual for more information).

Program Name - The program name is the name of the task or command that was executing when the Enforcer hit occurred.

Hunk Number and Offset - If possible, *Enforcer* also provides a hunk offset to where the program counter was if the hit occurred within the program's own code instructions. This will only work if the errant program was started from a shell.

Sample Mungwall Output

When *Mungwall* reports a hit, it lists:

The function that triggered the hit (either `AllocMem()` or `FreeMem()`). It includes the functions arguments.

The name of the offending program (''attempted by '<program name>' '').

The address of the offending program's Task structure (''at 0x<task address>'').

The address from which `AllocMem()/FreeMem()` was called. *Mungwall* reports two addresses: one labelled ''A:'' and one labelled ''C:''. The ''A:'' address is the address if `AllocMem()/FreeMem()` was called directly (i.e., using assembler or #pragmas) by the offending program. The ''C:'' address is the address if `AllocMem()/FreeMem()` was called from an *amiga.lib* C stub by the offending program. Since *Mungwall* patches the memory allocation functions, it can only guess the caller's address based on the return address it finds on the stack.

The stack pointer at the time of the Mungwall hit. It is labelled "SP:".

Note that *Mungwall* ignores the layers.library's partial deallocations. If any other debugging tools patch `AllocMem()` or `FreeMem()`, *Mungwall's* "A:" and "C:" addresses may be thrown off by additional information pushed on the stack, and *Mungwall* will also be unable to screen out the layers.library's partial deallocations (which will often show up as Mungwall hits on your task, CON:, or *input.device*).

Here are some sample Mungwall hits that were produced by a program called *mungwalltest*. As a reminder, the arguments for memory functions are `AllocMem(size,type)` and `FreeMem(address,size)`.

```
AllocMem(0x0,10000) attempted by 'mungwalltest' (at 0x339590)
  from A:0x35C03A C:0x35677E SP:0x35CFC0
```

Tried to allocate 0 bytes of memory.

```
FreeMem(0x0,16) attempted by 'mungwalltest' (at 0x339590)
  from A:0x35C068 C:0x3567C4 SP:0x35CFB8
```

Tried to free memory with a `NULL` pointer.

```
FreeMem(0x33BD10,0) attempted by 'mungwalltest' (at 0x339590)
  from A:0x35C068 C:0x3567D4 SP:0x35CFB0
```

Tried to free 0 bytes of memory.

```
Mis-aligned FreeMem(0x33BD14,16) attempted by 'mungwalltest' (at 0x339590)
      from A:0x35C068 C:0x3567E2 SP:0x35CFA8
```

Deallocation address is incorrect because it is not aligned according to AllocMem()'s lowest memory chunk size.

```
Mismatched FreeMem size 14!
Original allocation: 16 bytes from A:0x35C03A C:0x3567A0 Task 0x339590
Testing with original size.
```

Deallocation size does not match the allocation size.

```
19 byte(s) before allocation at 0x33BD10, size 16 were hit!
>$: BBBBBBBB BBBBBBBB BB536572 6765616E 74277320 50657070 65722000
```

Something wrote to the bytes which precede the allocation.

```
8 byte(s) after allocation at 0x33BD10, size 16 were hit!
>$: 75622042 616E6400 BBBBBBBB BBBBBBBB BBBBBBBB BBBBBBBB BBBBBBBB
```

Something wrote to the bytes which follow the allocation.

Using Enforcer and Mungwall Together

Alone, *Mungwall* can catch a large variety of memory-related software problems. If *Mungwall* and *Enforcer* are used together, they can catch many more well-hidden bugs. One bug that is hard to catch is when a program either mistakenly reads memory that does not belong to the program or reads its own memory without initializing the memory first.

These bugs are normally hard to catch because the behavior of the errant program usually depends on how it reacts to the data that happens to be stored in that memory. This makes the behavior of the bug erratic. Sometimes the errant program may crash. Sometimes the errant program may write over another program's data, causing the other program to crash. At other times, no noticeable abnormality takes place.

Mungwall and *Enforcer* together can help find these bugs. Because *Mungwall* fills freed memory with the same odd 32-bit values, when an errant program mistakenly accesses memory, the behavior of the bug will be consistent while *Mungwall* is running. Also, the values *Mungwall* puts in memory are more likely to cause the errant program to access Enforcer-protected memory, triggering an Enforcer hit.

A Sample Debugging Session

The following is an example of debugging an Enforcer hit that occurred using a test program called *ownertest*. This hit was generated on an A2500 with a 2.04 ROM image loaded using *ZKick*:

```
Program Counter (approximate)=  201946          Fault address       =       0
User stack pointer           =  566110          DOS process address =  38E888
Data: 00282D90 00000000 000003ED 0038FD8C 00000001 00000001 000E203B 00000001
Addr: 00225469 00000001 00282DE0 00448A3A 004487C0 004487CC 00001420 00002E28
Stck: 00448A3A 00223BA2 00280810 000003ED 0038FD8C 00000001 00000001 000E203B
READ-BYTE  (---)(F)(-)    SR=0010   SSW=0751
Background CLI, "ownertest"
```

The Program Counter is at $201946. On a machine with the Kickstart in ROM, the ROM addresses range from $F80000 to $FFFFFF under 2.0 or greater and from $FC0000 to $FFFFFF under 1.3. On a softkicked A3000, the addresses are in the same range as real ROM addresses. On a softkicked A2500, $201946 is a ROM address (in this case, the ROM ranges from $200000 to $27FFFF). The first thing to do is figure out in which ROM module the Enforcer hit occurred. The debugging tool *owner* , by Michael Sinz, will figure that out:

```
1.Ram Disk:> owner 0x201946

Address  - Owner
--------   -----
00201946 - in resident module: exec 37.132 (23.5.91)
```

Note that *Owner* looks at the ROM addresses of the Amiga on which it is executing, so you must run *owner* on the machine that generated the Enforcer hit.

Next, use the debugging tool *lvo* to figure out what function entry in the Exec ROM module is closest to that ROM address. Like *owner*, *lvo* also looks at local machine's ROM addresses, so you have to run *owner* on the machine that generated the Enforcer hit. Note that *lvo* requires the FD files to be in an "FD:" assign directory. Pass the address and the module name (from *owner*) to *lvo*.

```
1.Ram Disk:> lvo exec romaddress=0x201946

Closest to $201946 without going over:
exec.library  LVO $feec -276  FindName() jumps to $20192a on this system
```

Hmmm. A lot of functions use `FindName()` on Exec lists, so, in this case, the Program Counter does not pinpoint the problem. However, it does hint that `FindName()` probably received a bad string pointer. The "`READ-BYTE`" attribute of the Enforcer hit is an extra clue that the *Ownertest* program has a problem with a string pointer, since the most common `READ-BYTE` actions are on strings.

Let's check the Enforcer stack dump and see if there are any other ROM addresses there which might have called `FindName()`:

```
Stck: 00448A3A 00223BA2 00280810 000003ED 0038FD8C 00000001 00000001 000E203B
```

The address closest to the top of the stack that looks like a ROM address on this particular system is 0x00223BA2. Let's see what ROM module contains this address:

```
1.Ram Disk:> owner 0x223BA2

Address  - Owner
--------   -----
00223BA2 - in resident module: graphics 37.35 (23.5.91)
```

Let's see what function in the Graphics library is closest to this ROM address:

```
1.Ram Disk:> lvo graphics 0x223BA2

Closest to $223ba2 without going over:
graphics.library  LVO $ffb8 -72  OpenFont() jumps to $223b84 on this system
```

It looks like the *ownertest* program has a problem with a font name that was not properly initialized.

More Remote Debugging Tips

Debugging with *Enforcer* and *Mungwall* is even more effective when an application sends other debugging information to the serial or parallel port. The linker library *debug.lib* contains a `printf()`-like function, `kprintf()`, to print information to the serial port. The linker library *ddebug.lib* contains a similar function called `dprintf()` that prints debugging information to the parallel port. The output from these functions intermix with the output from *Enforcer* and *Mungwall*, making it easy to pinpoint which part of the code is causing *Enforcer* or Mungwall hits.

Functions like `kprintf()` and `dprintf()` are useful, but adding and removing them from programs can be tedious. One easy way to deal with this problem is to include them only when a special label is defined:

```
/**********    debug macros     ***********/
#define MYDEBUG  1
void kprintf(UBYTE *fmt,...);
void dprintf(UBYTE *fmt,...);
#define DEBTIME 0
#define bug printf
#if MYDEBUG
#define D(x) (x); if(DEBTIME>0) Delay(DEBTIME);
#else
#define D(x) ;
#endif /* MYDEBUG */
/********** end of debug macros **********/
```

Set `MYDEBUG` to 1 to turn on debugging. Set "bug" to:

"printf" to send debugging information to the default console,
"kprintf" to send debugging information to the serial port (link with *debug.lib*), and
"dprintf" to send debugging information to the parallel port (link with *ddebug.lib*).

When using this macro, make sure there two close parentheses before the semicolon at the end of each `D(bug())` statement.

Example macro usage:

```
win = OpenWindow(&mynewwin);
D(bug("Opened window at $%lx\n", win));
```

A different low-level method of figuring out which instructions caused an Enforcer hit is to disassemble program memory where the hit occurred. First, match the disassembly with your own code. Assembly programmers could just compare the disassembly to their source. Others could take the hex values of a sequence of position-independent 68000 instructions near the hit (i.e. no addresses except for offsets and branches) and do a search for this pattern in your object modules. If you find the pattern, do a mixed source and object disassembly of that object module and then look in the output for instructions matching those where the hit occurred. For example, with SAS's *OMD* you could compile your code with the flag -d1, then do the following:

```
1.Ram Disk:> OMD >ram:dump mymodule.o mymodule.c
```

Who Should Use Enforcer and Mungwall

If you are developing Amiga software, it is extremely important that you invest in a MMU, or at the very least make sure that your software is tested on machines with *Enforcer* and *Mungwall* (also test with *Enforcer* alone as *Mungwall* can hide certain types of bugs). If you are programming in assembly, you should also test with *Scratch* by Bill Hawes to catch improper usage of CPU registers.

Enforcer and *Mungwall* are not just for developers and QA departments. Anyone who uses software can help find bugs in it with *Enforcer*. During normal usage, they can catch hidden software problems. Many people at Commodore run *Enforcer* all of the time. As more and more people begin running these tools, they will become less tolerant of software that causes Enforcer and Mungwall hits.

At a small developer meeting at a recent Amiga trade show, CATS was disappointed to discover that, although the majority of the audience believed that they needed *Enforcer*, a relatively small percentage of them owned the equipment necessary to run it (i.e., an MMU). *If you don't have an MMU, get one.* The investment in an A3000, 68030 card, or 68020+MMU card will quickly pay for itself. It significantly cuts down development time because it quickly catches bugs that are otherwise hard to track down.

Amiga Mail
Volume II
May/June 1992

SANA-II Network Device Driver Specification

Amiga Networking Group
Brian Jackson, Dale Larson, Greg Miller, and Kenneth Dyke

The SANA-II Network Device Driver Specification is a standard for an Amiga software interface between networking hardware and network protocol stacks (or for software tools such as network monitors). A network protocol stack is a layer of software that network applications use to address particular processes on remote machines and to send data reliably in spite of hardware errors. There are several common network protocol stacks including TCP/IP, OSI, AppleTalk, DECNet and Novell.

SANA-II device drivers are intended to allow multiple network protocol stacks running on the same machine to share one network device. For example, the TCP/IP and AppleTalk protocol stacks could both run on the same machine over one ethernet board. The device drivers are also intended to allow network protocol stacks to be written in a hardware-independent fashion so that a different version of each protocol stack doesn't have to be written for each networking hardware device.

The standard does not address the writing of network applications. Application writers must not use SANA-II Device Drivers directly. Network applications must use the API provided by the network protocol software the application supports. There is not an Amiga standard network API at the time of this writing, though there is the AS225 TCP/IP package and its *socket.library* as well as other (third-party) packages.

To write a SANA-II device driver, you will need to be familiar with the specification documents for the hardware you are writing to and with the SANA-II Network Device Driver Specification.

To write a network protocol stack which will use SANA-II device drivers, you should have general familiarity with common network hardware and must be very familiar with the SANA-II Network Device Driver Specification as well as the specification for the protocol you are developing. If you are creating a new protocol, you must obtain a protocol type number for any hardware on which your protocol will be used.

Commodore supports the SANA-II specification by providing drivers for the Commodore-Amiga network hardware. We have an *A2065.device* (Ethernet) and intend to produce an *A2060.device* (ARCNET). We also try to examine review copies of third-party SANA-II networking hardware and software to try to make sure that they interoperate with our products.

This standard has undergone several drafts with long periods for comment from developers and the Amiga community at large. These drafts include a UseNet release which was also distributed on the Fish Disks in June, 1991 (as well as published in the '91 DevCon notes), and the November 7 Draft for Final Comment and Approval distributed via Bix, ADSP and UseNet. There were also several intermediate drafts with more limited distribution.

This version of the specification is final. Any new version of the standard (i.e., to add new features) is planned to be backward compatible. No SANA-II device driver or software utilizing those drivers should be written to any earlier version of the specification.

Distribution of this version of the standard is unlimited. Anyone may write Amiga software which implements a SANA-II network device driver or which calls a SANA-II network device driver without restriction and may freely distribute such software that they have written. Amiga is a registered trademark of Commodore-Amiga, Inc. Ethernet is a trademark of Xerox Corporation. ARCNET is a trademark of Datapoint Corporation. DECNet is a trademark of Digital Equipment Corporation. AppleTalk is a trademark of Apple Computer, Inc.

It is important to try to test each SANA-II device driver against all software which uses SANA-II devices. Available example programs are valuable in initial testing. The Amiga Networking Group is interested in receiving evaluation and/or beta test copies of all Amiga networking hardware, SANA-II device drivers and software which uses SANA-II devices. However, we make no assurances regarding any testing which we may or may not perform with such evaluation copies. Contact:

Amiga Networking Group
Commodore International Services Corporation
Technology Group
1200 Wilson Drive
West Chester, PA 19380, USA

Driver Form

SANA-II device drivers are Amiga Exec device drivers. They use an extended IORequest structure and a number of extended commands for tallying network statistics, sending broadcasts and multicasts, network addressing and the handling of unexpected packets. The *Amiga ROM Kernel Reference Manual: Devices* includes information on how to construct an Exec device.

Let's check the Enforcer stack dump and see if there are any other ROM addresses there which might have called `FindName()`:

```
Stck: 00448A3A 00223BA2 00280810 000003ED 0038FD8C 00000001 00000001 000E203B
```

The address closest to the top of the stack that looks like a ROM address on this particular system is 0x00223BA2. Let's see what ROM module contains this address:

```
1.Ram Disk:> owner 0x223BA2

Address  - Owner
--------   -----
00223BA2 - in resident module: graphics 37.35 (23.5.91)
```

Let's see what function in the Graphics library is closest to this ROM address:

```
1.Ram Disk:> lvo graphics 0x223BA2

Closest to $223ba2 without going over:
graphics.library  LVO $ffb8 -72  OpenFont() jumps to $223b84 on this system
```

It looks like the *ownertest* program has a problem with a font name that was not properly initialized.

More Remote Debugging Tips

Debugging with *Enforcer* and *Mungwall* is even more effective when an application sends other debugging information to the serial or parallel port. The linker library *debug.lib* contains a `printf()`-like function, `kprintf()`, to print information to the serial port. The linker library *ddebug.lib* contains a similar function called `dprintf()` that prints debugging information to the parallel port. The output from these functions intermix with the output from *Enforcer* and *Mungwall*, making it easy to pinpoint which part of the code is causing *Enforcer* or Mungwall hits.

Functions like `kprintf()` and `dprintf()` are useful, but adding and removing them from programs can be tedious. One easy way to deal with this problem is to include them only when a special label is defined:

```
/**********    debug macros     ***********/
#define MYDEBUG  1
void kprintf(UBYTE *fmt,...);
void dprintf(UBYTE *fmt,...);
#define DEBTIME 0
#define bug printf
#if MYDEBUG
#define D(x) (x); if(DEBTIME>0) Delay(DEBTIME);
#else
#define D(x) ;
#endif /* MYDEBUG */
/********** end of debug macros **********/
```

Set `MYDEBUG` to 1 to turn on debugging. Set "bug" to:

"printf" to send debugging information to the default console,
"kprintf" to send debugging information to the serial port (link with *debug.lib*), and
"dprintf" to send debugging information to the parallel port (link with *ddebug.lib*).

When using this macro, make sure there two close parentheses before the semicolon at the end of each `D(bug())` statement.

Example macro usage:

```
win = OpenWindow(&mynewwin);
D(bug("Opened window at $%lx\n", win));
```

A different low-level method of figuring out which instructions caused an Enforcer hit is to disassemble program memory where the hit occurred. First, match the disassembly with your own code. Assembly programmers could just compare the disassembly to their source. Others could take the hex values of a sequence of position-independent 68000 instructions near the hit (i.e. no addresses except for offsets and branches) and do a search for this pattern in your object modules. If you find the pattern, do a mixed source and object disassembly of that object module and then look in the output for instructions matching those where the hit occurred. For example, with SAS's *OMD* you could compile your code with the flag -d1, then do the following:

```
1.Ram Disk:> OMD >ram:dump mymodule.o mymodule.c
```

Who Should Use Enforcer and Mungwall

If you are developing Amiga software, it is extremely important that you invest in a MMU, or at the very least make sure that your software is tested on machines with *Enforcer* and *Mungwall* (also test with *Enforcer* alone as *Mungwall* can hide certain types of bugs). If you are programming in assembly, you should also test with *Scratch* by Bill Hawes to catch improper usage of CPU registers.

Enforcer and *Mungwall* are not just for developers and QA departments. Anyone who uses software can help find bugs in it with *Enforcer*. During normal usage, they can catch hidden software problems. Many people at Commodore run *Enforcer* all of the time. As more and more people begin running these tools, they will become less tolerant of software that causes Enforcer and Mungwall hits.

At a small developer meeting at a recent Amiga trade show, CATS was disappointed to discover that, although the majority of the audience believed that they needed *Enforcer*, a relatively small percentage of them owned the equipment necessary to run it (i.e., an MMU). *If you don't have an MMU, get one.* The investment in an A3000, 68030 card, or 68020+MMU card will quickly pay for itself. It significantly cuts down development time because it quickly catches bugs that are otherwise hard to track down.

Amiga Mail
Volume II
May/June 1992

SANA-II Network Device Driver Specification

Amiga Networking Group
Brian Jackson, Dale Larson, Greg Miller, and Kenneth Dyke

The SANA-II Network Device Driver Specification is a standard for an Amiga software interface between networking hardware and network protocol stacks (or for software tools such as network monitors). A network protocol stack is a layer of software that network applications use to address particular processes on remote machines and to send data reliably in spite of hardware errors. There are several common network protocol stacks including TCP/IP, OSI, AppleTalk, DECNet and Novell.

SANA-II device drivers are intended to allow multiple network protocol stacks running on the same machine to share one network device. For example, the TCP/IP and AppleTalk protocol stacks could both run on the same machine over one ethernet board. The device drivers are also intended to allow network protocol stacks to be written in a hardware-independent fashion so that a different version of each protocol stack doesn't have to be written for each networking hardware device.

The standard does not address the writing of network applications. Application writers must not use SANA-II Device Drivers directly. Network applications must use the API provided by the network protocol software the application supports. There is not an Amiga standard network API at the time of this writing, though there is the AS225 TCP/IP package and its *socket.library* as well as other (third-party) packages.

To write a SANA-II device driver, you will need to be familiar with the specification documents for the hardware you are writing to and with the SANA-II Network Device Driver Specification.

To write a network protocol stack which will use SANA-II device drivers, you should have general familiarity with common network hardware and must be very familiar with the SANA-II Network Device Driver Specification as well as the specification for the protocol you are developing. If you are creating a new protocol, you must obtain a protocol type number for any hardware on which your protocol will be used.

Commodore supports the SANA-II specification by providing drivers for the Commodore-Amiga network hardware. We have an *A2065.device* (Ethernet) and intend to produce an *A2060.device* (ARCNET). We also try to examine review copies of third-party SANA-II networking hardware and software to try to make sure that they interoperate with our products.

This standard has undergone several drafts with long periods for comment from developers and the Amiga community at large. These drafts include a UseNet release which was also distributed on the Fish Disks in June, 1991 (as well as published in the '91 DevCon notes), and the November 7 Draft for Final Comment and Approval distributed via Bix, ADSP and UseNet. There were also several intermediate drafts with more limited distribution.

This version of the specification is final. Any new version of the standard (i.e., to add new features) is planned to be backward compatible. No SANA-II device driver or software utilizing those drivers should be written to any earlier version of the specification.

Distribution of this version of the standard is unlimited. Anyone may write Amiga software which implements a SANA-II network device driver or which calls a SANA-II network device driver without restriction and may freely distribute such software that they have written. Amiga is a registered trademark of Commodore-Amiga, Inc. Ethernet is a trademark of Xerox Corporation. ARCNET is a trademark of Datapoint Corporation. DECNet is a trademark of Digital Equipment Corporation. AppleTalk is a trademark of Apple Computer, Inc.

It is important to try to test each SANA-II device driver against all software which uses SANA-II devices. Available example programs are valuable in initial testing. The Amiga Networking Group is interested in receiving evaluation and/or beta test copies of all Amiga networking hardware, SANA-II device drivers and software which uses SANA-II devices. However, we make no assurances regarding any testing which we may or may not perform with such evaluation copies. Contact:

Amiga Networking Group
Commodore International Services Corporation
Technology Group
1200 Wilson Drive
West Chester, PA 19380, USA

Driver Form

SANA-II device drivers are Amiga Exec device drivers. They use an extended IORequest structure and a number of extended commands for tallying network statistics, sending broadcasts and multicasts, network addressing and the handling of unexpected packets. The *Amiga ROM Kernel Reference Manual: Devices* includes information on how to construct an Exec device.

Opening a SANA-II Device

As when opening any other Exec device, on the call to `OpenDevice()` a SANA-II device receives an IORequest structure which the device initializes for the opener's use. The opener must copy this structure if it desires to use multiple asynchronous requests. The SANA-II IORequest is defined as follows:

```
struct IOSana2Req
{
    struct IORequest ios2_Req;
    ULONG ios2_WireError;
    ULONG ios2_PacketType;
    UBYTE ios2_SrcAddr[SANA2_MAX_ADDR_BYTES];
    UBYTE ios2_DstAddr[SANA2_MAX_ADDR_BYTES];
    ULONG ios2_DataLength;
    APTR *ios2_Data;
    APTR *ios2_StatData;
    APTR *ios2_BufferManagement;
};
```

ios2_Req - A standard Exec device IORequest.

ios2_WireError - A more specific device code which may be set when there is an io_Error. See *<devices/sana2.h>* for the defined WireErrors.

ios2_PacketType - The type of packet requested. See the section on "Packet Types".

ios2_SrcAddr - The device fills in this field with the interface (network hardware) address of the source of the packet that satisfied a read command. The bytes used to hold the address will be left justified but the bit layout is dependent on the particular type of network.

ios2_DstAddr - Before the device user sends a packet, it fills this with the interface destination address of the packet. On receives, the device fills this with the interface destination address. Other commands may use this field differently (see the SANA-II Network Device Driver Autodocs). The bytes used to hold the address will be left justified but the bit layout is dependent on the particular type of network.

ios2_DataLength - The device user initializes this field with the amount of data available in the Data buffer before passing the IOSana2Req to the device. The device fills in this field with the size of the packet data as it was sent on the wire. This does not include the header and trailer information. Depending on the network type and protocol type, the driver may have to calculate this value. This is generally used only for reads and writes (including broadcast and multicast).

ios2_Data - A pointer to some abstract data structure containing packet data. Drivers may not directly manipulate or examine anything pointed to by Data! This is generally used only for reads and writes (including broadcast and multicast).

ios2_StatData - Pointer to a structure in which to place a snapshot of device statistics. The data area must be long word aligned. This is only used on calls to the statistics commands.

ios2_BufferManagement - The opener places a pointer to a tag list in this field before calling `OpenDevice()`. Functions pointed to in the tag list are called by the device when processing IORequests from the opener. When returned from `OpenDevice()`, this field contains a pointer to driver-private information used to access these functions. See "Buffer Management" below for more details.

The flags used with the device on `OpenDevice()` are (`SANA2OPB_xxx`):

`SANA2OPB_MINE` - Exclusive access to the unit requested.

`SANA2OPB_PROM` - Promiscuous mode requested. Hardware which supports promiscuous mode allows all packets sent over the wire to be captured whether or not they are addressed to this node.

The flags used during I/O requests are (`SANA2IOB_xxx`):

`SANA2IOB_RAW` - Raw packet read/write requested. Raw packets should include the entire data-link layer packet. Devices with the same hardware device number should have the same raw packet format.

`SANA2IOB_BCAST` - Broadcast packet (received).

`SANA2IOB_MCAST` - Multicast packet (received).

`SANA2IOB_QUICK` - Quick IO requested.

Buffer Management

Unlike most other Exec Device drivers, SANA-II drivers have no internal buffers. Instead, they read/write to/from an abstract data structure allocated by the driver user. The driver accesses these buffers only via functions that the driver user provides to the driver. The driver user must provide two functions--one copies data to the abstract data structure and one copies data from the abstract data structure. The driver user can therefore choose the data structure used for buffer management by both the driver and driver user in order to have efficient memory and CPU usage overall.

The IOSana2Req contains a pointer to data and the length of said data. A driver is not allowed to make assumptions about how the data is stored. The driver cannot directly manipulate or examine the buffer in any manner. The driver can only access the buffer by calling the functions provided by the driver user.

Before calling `OpenDevice()`, the driver user points ios2_BufferManagement to a list of tags (defined in *<devices/sana2.h>*) which include pointers to the buffer management functions required by the driver (defined below). The driver will fail to open if the driver user does not supply all of the required functions. If the device opens successfully, the driver sets ios2_BufferManagement to a value which this opener must use in all future calls to the driver. This "magic cookie" is used from then on to access these functions (a "magic cookie" is a value which one software entity passes to

another but which is only meaningful to one of the software entities). The driver user may not use the "magic cookie" in any way--it is for the driver to do with as it wishes. The driver could in theory choose to just copy the tag list to driver-owned memory and then parse the list for every IORequest, but it is much more efficient for the driver to create some sort of table of functions and to point ios2_BufferManagement to that table.

The specification currently includes only two tags for the `OpenDevice()` ios2_BufferManagement tag list:

`S2_CopyToBuff` - This is a pointer to a function which conforms to the CopyToBuff Autodoc.
`S2_CopyFromBuff` - This is a pointer to a function which conforms to the CopyFromBuff Autodoc.

Packet Type

Network frames always have a type field associated with them. These type fields vary in length, position and meaning by frame type (frame types generally correspond one-to-one with hardware types, but see "Ethernet Packet Types" below). The meanings of the type numbers are always carefully defined and every type number is registered with some official body. Do not use a type number which is not registered for any standard hardware you use or in a manner inconsistent with that registration.

The type field allows the SANA-II device driver to fulfill `CMD_READs` based on the type of packet the driver user wants. Multiple protocols can therefore run over the same wire using the same driver without stepping on each other's toes.

Packet types are specified as a long word. Unfortunately, the type field means different things on different wires. Driver users must allow their software to be configured with a SANA-II device name, unit number and the type number(s) used by the protocol stack with each device. This way, if new hardware becomes available, a hardware manufacturer can supply a listing of type assignments to configure pre-existing software.

Ethernet Packet Types

Ethernet has a special problem with packet types. Two types of ethernet frames can be sent over the same wire--ethernet and 802.3. These frames differ in that the Type field of an ethernet frame is the Length field of an 802.3 frame. This creates a problem in that demultiplexing incoming packets can be cumbersome and inefficient, as well as requiring driver users to be aware of the frame type used.

All 802.3 frames have numbers less than 1500 in the Type field. The only frames with numbers less than 1500 in the type field are 802.3 frames. SANA-II ethernet drivers abnormally return packets

contained in ethernet frames when the requested Type falls within the 802.3 range--if the Type requested is within the 802.3 range, the driver returns the next packet contained within an 802.3 frame, regardless of the type specified for the packet within the 802.3 frame. This requires that there be no more than one driver user requesting 802.3 packets and that it do its own interpretation of the frames.

ARCNET Frames

ARCNET also has a special problem with framing. ARCNET frames consist of a hardware header and a software header. The software header is in the data area of the hardware packet, and includes at least the protocol ID.

There are two types of software header. Old-style ARCNET software headers consist entirely of a one or two byte protocol ID. New ARCNET software headers (defined in RFC 1201 and in the paper "ARCNET Packet Header Definition Standard", Novell, Inc., 1989) include more information. They allow more efficient use of ARCNET through data link layer fragmentation and reassembly (ARCNET has a small Maximum Transmission Unit) and allow sending any size packet up to the MTU (rather than requiring that packets of size 253, 254 and 255 be padded to at least 256 bytes).

SANA-II device drivers for ARCNET should implement the old ARCNET packet headers. Driver users which wish to interoperate with platforms using the new software headers must add the new fields to the data to be sent and must process it for incoming data. A SANA-II driver which implemented the data link layer fragmentation internally (and advertised a large MTU) could be more efficient than requiring the driver user to do it. This would make driver writing more difficult and reduce interoperability, but if there is ever a demand for that extra performance, a new hardware type may be assigned by Commodore for SANA-II ARCNET device drivers which implement the new framing.

Addressing

In the SANA-II standard, network hardware addresses are stored in an array of n bytes. No meaning is ascribed by the standard to the contents of the array.

In case there exists a network which does not have an address field consisting of a number of bits not divisible by eight, add pad bits at the end of the bit stream. For example, if an address is ten bits long it will be stored like this:

```
98765432 10PPPPPP
BYTE 0   BYTE 1
```

Where the numerals are bit numbers and 'P' is a pad (ignored) bit.

Driver users which do not implement the bit shifting necessary to use a network with such addressing (if one exists) should at least check the number of significant bits in the address field (returned from the device's S2_DEVICEQUERY function) to make sure that it is evenly divisible by eight.

Driver users will map hardware addresses to protocol addresses in a protocol and hardware dependent manner, as described by the relevant standards (i.e., RFC 826 for TCP/IP over Ethernet, RFC 1201 or 1051 for TCP/IP over ARCNET). Some protocols will always use the same mapping on all hardware, but other protocols will have particular address mapping schemes for some particular hardware and a reasonable default for other (unknown) hardware.

Some SANA-II devices will have "hardware addresses" which aren't really hardware addresses. As an example, consider PPP (Point-to-Point Protocol). PPP is a standard for transmitting IP packets over a serial line. It uses IP addresses negotiated during the establishment of a connection. In a SANA-II driver implementation of PPP, the driver would negotiate the address at S2_CONFIGINTERFACE. Thus, the address in SrcAddr returned by the device on an S2_CONFIGINTERFACE (or in a subsequent S2_GETSTATIONADDRESS) will be a protocol address, not a true hardware address.

Note: Some hardware always uses a ROM hardware address. Other hardware which has a ROM address or is configurable with DIP switches may be overridden by software. Some hardware always dynamically allocates a new hardware address at initialization. See "Configuration" for details on how this is handled by driver writers and by driver users.

Hardware Type

The HardwareType returned by the device's S2_DEVICEQUERY function is necessary for those protocols whose standards require different behavior on different hardware. It is also useful for determining appropriate packet type numbers to use with the device. The HardwareType values already issued for standard network hardware are the same as those in RFC 1060 (assigned numbers). Hardware developers implementing networks without a SANA-II hardware number must contact CATS to have a new hardware type number assigned. Driver users should all have reasonable defaults which can be used for hardware with which they are not familiar.

Errors

The SANA-II extended IORequest structure (struct IOSana2Req) includes both the ios2_Error and ios2_WireError fields. Driver users must always check IOSana2Reqs on return for an error in ios2_Error. ios2_Error will be zero if no error occurred, otherwise it will contain a value from *<exec/errors.h>* or *<devices/sana2.h>*. If there was an error, there may be more specific information in ios2_WireError. Drivers are required to fill in the WireError if there is an applicable error code.

Stats

S2_TRACKTYPE	S2_GETTYPESTATS	S2_GETGLOBALSTATS
S2_UNTRACKTYPE	S2_GETSPECIALSTATS	S2_READORPHAN

There are many statistics which may be very important to someone trying to debug, tune or optimize a protocol stack, as well as to the end user who may need to tune parameters or investigate a problem. Some of these statistics can only be kept by the SANA-II driver, thus there are several required and optional statistics and commands for this purpose.

S2_TRACKTYPE tells the device driver to gather statistics for a particular packet type. S2_UNTRACKTYPE tells it to stop (keeping statistics by type causes the driver to use additional resources). S2_GETTYPESTATS returns any statistics accumulated by the driver for a type being tracked (stats are lost when a type is S2_UNTRACKTYPE'd). Drivers are required to implement the functionality of type tracking. The stats are returned in a struct Sana2PacketTypeStats:

```
struct Sana2PacketTypeStats
{
    ULONG PacketsSent;
    ULONG PacketsReceived;
    ULONG BytesSent;
    ULONG BytesReceived;
    ULONG PacketsDropped;
};
```

PacketsSent - Number of packets of a particular type sent.
PacketsReceived - Number of packets of a particular type that satisfied a read command.
BytesSent - Number of bytes of data sent in packets of a particular type.
BytesReceived - Number of bytes of data of a particular packet type that satisfied a read command.
PacketsDropped - Number of packets of a particular type that were received while there were no pending reads of that packet type.

S2_GETGLOBALSTATS returns global statistics kept by the driver. Drivers are required to keep all applicable statistics. Since all are applicable to most hardware, most drivers will maintain all statistics. The stats are returned in a struct Sana2DeviceStats:

```
struct Sana2DeviceStats
{
    ULONG PacketsReceived;
    ULONG PacketsSent;
    ULONG BadData;
    ULONG Overruns;
    ULONG UnknownTypesReceived;
    ULONG Reconfigurations;
    struct timeval LastStart;
};
```

PacketsReceived - Number of packets that this unit has received.
PacketsSent - Number of packets that this unit has sent.
BadData - Number of bad packets received (i.e., hardware CRC failed).
Overruns - Number of packets dropped due to insufficient resources available in the network interface.
UnknownTypeReceived - Number of packets received that had no pending read command with the appropriate packet type.
Reconfigurations - Number of network reconfigurations since this unit was last configured.
LastStart - The time when this unit last went on-line.

`S2_GETSPECIALSTATS` returns any special statistics kept by a particular driver. Each new wire type will have a set of documented, required statistics for that wire type and a standard set of optional statistics for that wire type (optional because they might not be available from all hardware). The data returned by `S2_GETSPECIALSTATS` will require wire-specific interpretation. See *<devices/sana2specialstats.h>* for currently defined special statistics. The statistics are returned in the following structures:

```
struct Sana2SpecialStatRecord
{
    ULONG Type;
    ULONG Count;
    char *String;
};
```

Type - Statistic identifier.
Count - Statistic itself.
String - An identifying, null-terminated string for the statistic. Should be plain ASCII with no formatting characters.

```
struct Sana2SpecialStatHeader
{
    ULONG RecordCountMax;
    ULONG RecordCountSupplied;
    struct Sana2SpecialStatRecord[RecordCountMax];
};
```

RecordCountMax - There is space for this many records into which statistics may be placed.
RecordCountSupplied - Number of statistic records supplied.

`S2_READORPHAN` is not, strictly speaking, a statistical function. It is a request to read any packet of a type for which there is no outstanding `CMD_READ`. `S2_READORPHAN` might be used in the same manner as many statistics, though, such as to determine what packet types are causing overruns, etc.

Configuration

S2_DEVICEQUERY S2_CONFIGINTERFACE S2_GETSTATIONADDRESS

The device driver needs to configure the hardware before using it. The driver user must know some network hardware parameters (hardware address and MTU, for example) when using it. These commands address those needs.

When a driver user is initialized, it should try to S2_CONFIGINTERFACE even though an interface can only be configured once and someone else may have done it. Before you call S2_CONFIGINTERFACE, first call S2_GETSTATIONADDRESS to determine the factory address (if any). Also provide for user-override of the factory address (that address may be optional and the user may need to override it). When S2_CONFIGINTERFACE returns, check the ios2_SrcAddr for the actual address the hardware has been configured with. This is because some hardware (or serial line standards such as PPP) always dynamically allocates an address at initialization.

Driver users will want to use S2_DEVICEQUERY to determine the MTU and other characteristics of the network. The structure returned from S2_DEVICEQUERY is defined as:

```
struct Sana2DeviceQuery
{
    ULONG SizeAvailable;
    ULONG SizeSupplied;
    ULONG DevQueryFormat;
    ULONG DeviceLevel;
    UWORD AddrFieldSize;
    ULONG MTU;
    ULONG BPS;
    ULONG HardwareType;
};
```

SizeAvailable - Size, in bytes, of the space available in which to place device information. This includes both size fields.

SizeSupplied - Size, in bytes, of the data supplied.

DevQueryFormat - The format defined here is format 0.

DeviceLevel - This spec defines level 0.

AddrFieldSize - The number of bits in an interface address.

MTU - Maximum Transmission Unit, the size, in bytes, of the maximum packet size, not including header and trailer information.

BPS - Best guess at the raw line rate for this network in bits per second.

HardwareType - Specifies the type of network hardware the driver controls.

On-line

S2_ONLINE S2_ONEVENT S2_OFFLINE

In order to run hardware tests on an otherwise live system, the S2_OFFLINE command allows the SANA-II device driver to be "turned off" until the tests are complete and an ONLINE is sent to the driver. S2_ONLINE causes the interface to re-configure and re-initialize. Any packets destined for the hardware while the device is off-line will be lost. All pending and new requests to the driver shall be returned with S2ERR_OUTOFSERVICE when a device is off-line.

All driver users must understand that any IO request may return with S2ERR_OUTOFSERVICE because the driver is off-line (any other program may call S2_OFFLINE to make it so). In such an event, the driver will usually want to wait until the unit comes back on-line (for the program which called S2_OFFLINE to call S2_ONLINE). It may do this by calling S2_ONEVENT to wait for S2EVENT_ONLINE. S2_ONEVENT allows the driver user to wait on various events.

A driver must track events, but may not distinguish between some types of events. Drivers return S2_ONEVENT with S2_ERR_NOT SUPPORTED and S2WERR_BAD_EVENT for unsupported Events. One error may cause more than one Event (see below). Errors which seem to have been caused by a malformed or unusual request should not generally trigger an event.

Event types (S2EVENT_xxx):

ERROR - Return when any error occurs.
TX - Return on any transmit error (always an error).
RX - Return on any receive error (always an error).
ONLINE - Return when unit goes on-line or return immediately if unit is already on-line (not an error).
OFFLINE - Return when unit goes off-line or return immediately if unit is already off-line (not an error).
BUFF - Return on any buffer management function error (always an error).
HARDWARE - Return when any hardware error occurs (always an error, may be a TX or RX, too).
SOFTWARE - Return when any software error occurs (always an error, may be a TX or RX, too).

Acknowledgments

Many people and companies have contributed to the SANA-II Network Device Driver Specification. The original SANA-II Autodocs and includes were put together by Ray Brand, Perry Kivolowitz (ASDG) and Martin Hunt. Those original documents evolved to their current state and grew to include this document at the hands of Dale Larson and Greg Miller. Brian Jackson and John Orr provided valuable editing. Randell Jesup has provided sage advice on several occasions. The buffer

management callback mechanism was his idea. Dale Luck (GfxBase) and Rick Spanbauer (Ameristar Technologies) have provided valuable comments throughout the process. Nicolas Benezan (ADONIS) provided many detailed and useful comments on weaknesses in late drafts of the specification. Thanks to all the above and the numerous others who have contributed with their comments, questions and discussions.

Unresolved Issues

Unfortunately, it isn't possible to completely isolate network protocols from the hardware they run on. Hardware types and addressing both remain somewhat hardware-dependent in spite of our efforts. See the "Packet Type" section for an explanation of how packet types are handled and why protocols cannot be isolated from them. See the "Addressing" section for an explanation of how addressing is handled any why protocols cannot be isolated from it.

Additionally, there are at least two cases where a hardware type has multiple framing methods in use (ethernet/802.3 and arcnet/(Novell) "ARCNET Packet Header Definition Standard"). In both cases, software which must interoperate with other platforms on this hardware may need to be aware of the distinctions and may have to do extra processing in order to use the appropriate frame type. See the sections on "Ethernet Packet Types" and on "ARCNET frames" for more details.

Amiga Mail
Volume II
May/June 1992

Four Can Play–Supporting Parallel Port Joysticks

by Chris Ludwig

One of the many reasons people buy computer-based products is to use them as an "entertainment box". Amiga users are no exception in this regard. The Amiga has all the ingredients needed to be a great game machine; fast high-quality graphics, high-fidelity stereo sound, fast processor, and built-in joystick ports.

With this level of hardware support, it's no wonder that some of the industry's best games continue to be written for the Amiga first.

The only kind of games that don't currently lend themselves to the Amiga's built-in hardware are those that are meant to be played by more than two players simultaneously. Several types of games fall into this category, the most notable being the scrolling cooperative. This type of game, where up to four players control on-screen versions of themselves, is quite popular in video arcades. Another format that lends itself to having more than two joysticks is the board game simulation. It's simply easier to play a four player board game if each player has his or her own control.

Because the Amiga has only two joystick ports, there are currently only a small number of "greater than two" player games available. So how do these games get around the two port limitation? The developers of these games have worked up a little bit of external hardware that remaps some parallel port input lines as two more joystick ports for single button joysticks.

Fortunately, these developers are all using the same wiring in their interfaces. Unfortunately, neither the interface nor its specifications have been widely available.

This article explains the construction of these parallel port joystick adapters (this article refers to them "+2" adapters), and demonstrates a coding strategy which will allow your software to access these two new ports. Though the instructions are fairly simple, if not done properly, it is possible to damage your computer, so Commodore does not recommend that you build these yourself. This information is provided for your convenience and is expressly subject to the disclaimer and warnings found at the beginning of this publication.

Parallel function	Parallel port (DB-25 male)	JOY3 (DB-9)	JOY4 (DB-9)	Joystick function
Data bit 0	2	1		JOY3 UP
Data bit 1	3	2		JOY3 DOWN
Data bit 2	4	3		JOY3 LEFT
Data bit 3	5	4		JOY3 RIGHT
Data bit 4	6		1	JOY4 UP
Data bit 5	7		2	JOY4 DOWN
Data bit 6	8		3	JOY4 LEFT
Data bit 7	9		4	JOY4 RIGHT
Busy	11		6	JOY4 FIRE
Select (Online)	13	6		JOY3 FIRE
ground	18		8	JOY4 GROUND
ground	19	8		JOY3 GROUND

The "+2" Interface

Building the "+2" Interface

The interface that these two extra joystick ports require is quite straightforward in design and does not require any power. Essentially, two male 9-pin "d-sub" connectors (the type commonly used for joystick ports) are wired directly to a 25-pin "d-sub" connector.

The gender of the 25-pin connector depends on the Amiga model. Amiga 1000 computers have a non-standard, male 25-pin parallel port, so the Amiga 1000 adapters will need a female connector. All other Amigas have standard female 25-pin parallel ports, so adapters for A500, A600, A2000, A3000, A3000T, and CDTV units should have a male 25-pin connector.

The only difference between the requirements for the A1000 "+2" adapter and the "+2" adapter for the rest of the Amiga/CDTV product line is the gender. The "+2" adapter only uses parallel port pins that all Amigas/CDTVs have in common. These particular pins were chosen so that one "+2" adapter and a gender changer could connect two joysticks to any existing Amiga. Because some of the unused pins carry voltages, *only wire the pins specified in the chart! Doing otherwise can destroy components inside the Amiga!*

The "+2 Interface" chart above contains a list of the pins which need to be connected to create the "+2" adapter.

Parallel function	Parallel port (DB-25 male)	JOY3 (DB-25 male)	JOY4 (DB-25 male)	Joystick function
Data bit 0	2	1		JOY3 UP
Data bit 1	3	2		JOY3 DOWN
Data bit 2	4	3		JOY3 LEFT
Data bit 3	5	4		JOY3 RIGHT
Data bit 4	6		9	JOY4 UP
Data bit 5	7		10	JOY4 DOWN
Data bit 6	8		11	JOY4 LEFT
Data bit 7	9		12	JOY4 RIGHT
Busy	11		22	JOY4 FIRE
Select (Online)	13	14		JOY3 FIRE
ground	18		24	JOY4 GROUND
ground	19	16		JOY3 GROUND

The Non-Production "+2" Interface

Housing the Interface

For development purposes, you can construct a "+2" adapter from two 25-pin male d-sub connectors. One connector attaches to the parallel port, while the other accepts the two 9-pin female joystick plugs. Simply build the "number 3" joystick connector into the leftmost 9 pins, and the "number 4" connector into the rightmost 9 pins. The pin-outs for this setup are in the "Non-Production +2 Interface" chart above.

This arrangement is attractive to developers primarily because it has few parts and it's easy to construct. It is appropriate only for developer use and testing. Do not consider it for any sort of user distribution, because the connectors for certain brands of joysticks are too big to fit side by side into a 25-pin connector. Users will become quite irate upon finding that they can't use their "wiz-stik 5000" with their 4 player game adapter.

With this in mind, adapters intended for use by end users should be built into some sort of casework. The box should include two suitably spaced joystick connectors, and a cable that allows the user to easily attach the adapter to the parallel port. Be sure that the cable is long enough to allow users to place the adapter where they can get to it easily.

Alternatively, the device could simply be a pair of long cables in a "Y" shape, with a parallel connector at the base of the "Y", and joystick connector on the other ends. Such an adapter would probably be the cheapest way to go.

Developers must caution end users to turn their computer's power off before plugging in a "+2" adapter. This will prevent users from accidentally destroying the 8520 chips which control the parallel port. To further prevent 8520 damage, developers should choose 25-pin connectors without a metal case or shield. While plugging the device in, it's rather easy to accidentally brush (and short) the metal shields of certain connectors against the pins of the parallel port.

No matter what the decision regarding implementation, remember that game players can be quite rough. All cables should be suitably long, and all connectors should utilize a proper strain relief system.

Supporting the Interface in Software

Because this "+2" joystick adapter attaches to the parallel port, supporting it in software is simply a matter of reading the correct lines on the parallel port.

Many Amiga game programmers make the decision to throw away the OS and directly manipulate the hardware. These programmers will no doubt come up with their own personal scheme for reading the appropriate information from the port. Other, more forward thinking developers, may want to write their software in such a way as to be compatible with the OS.

The following pair of code samples work in tandem to demonstrate one method of reading the necessary information from the parallel port in an OS friendly manner. The C program *4play.c* is a simple example that demonstrates how to access the assembly functions. The assembly language program properly allocates the parallel port and any necessary signal lines, warning the OS not to let any other applications use them. Other assembly routines do the actual hardware level reading and place the joystick values into C variables. A cleanup routine releases the port and signal lines. The example does not mask out the directional or fire button bits, it only prints the raw joystick data. I'll leave it as an exercise for the reader to interpret the meaning of the raw data.

```
ASM    = asm
AFLAGS = -iinclude:
CC     = lc
CFLAGS = -cfistE
LN     = blink

LIBS = LIB:lc.lib LIB:amiga.lib

.c.o:
            $(CC) $(CFLAGS) $*.c

.asm.o:
            $(ASM) $(AFLAGS) $*.asm

4play: 4play.o read34.o
            $(LN) FROM LIB:c.o,4play.o,read34.o TO 4play LIBRARY $(LIBS)
```

```
/*
 * 4play.c
 */

#include    <exec/types.h>
#include    <libraries/dos.h>

#include    <stdlib.h>
#include    <stdio.h>

#include    <clib/exec_protos.h>

UBYTE portdata;
UBYTE *portptr = &portdata;

UBYTE firedata;
UBYTE *fireptr = &firedata;

extern int getport(void);
extern void read34(void);
extern void freeport(void);


/*
 * Lattice control-c stop...
 */
int CXBRK(void) { return(0); }      /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }   /* really */


void Quit(char whytext[], LONG return_code)
{
    if(return_code==0) freeport();  /* Assembly routine to
                                       de-allocate parallel port */

    printf("%s\n",whytext);

    exit(return_code);              /* returning non-zero
                                       terminates the program */
}


void main(void)
{
    BOOL done=FALSE;
    UBYTE error;

    /* getport() is an assembly routine that allocates the parallel port
     * and makes all the lines we're interested in "read" lines.
     */
    if(error=getport()) Quit("Parallel port in use",25);

    /* WARNING:
     * This example continuously reads the ports and checks for CTRL_C,
     * thereby eating a lot of CPU time.  Actual applications that expect
     * to be even more system friendly might want to set up some interrupts
     * on the fire button lines, such that the game can read the ports less
     * often, but never miss a "fire" press.
     */

    while(!done)
    {
        read34();       /* read34() is the assembly routine that copies the
                         * relavent data from the port into our variables.
                         */

        /* We'll just print the raw bytes from the read, and leave it as an
         * exercise for the reader to mask out the relevant bits.
         * (Check the pinouts to find which bits the switches appear at.)
         */
        printf("portdata = %u, firedata = %u\n",portdata,firedata);

        /* Check CTRL_C */
        if(SetSignal(0L,0L) & SIGBREAKF_CTRL_C) /* Hit since last check? */
        {
            SetSignal(0L,SIGBREAKF_CTRL_C); /* Clear old status */
            done=TRUE;
        }
    }
    Quit("Ctrl-C was pressed.",0);
}
```

```
;
; read34.asm
;
; interface code for the "2 more players" parallel port hack.
;
        csect text          ; this here's the meat

        xdef   Name         ; Name of our application, so that
Name    dc.b   '4play',0    ; other applications will know
                            ; who's tying up the port. ;-)


        xdef   _read34      ; function names for linker
        xdef   _getport
        xdef   _freeport

        xref   _portptr     ; c pointer for port data
        xref   _fireptr     ; c pointer for fire buttons

        xref   _SysBase     ; exec system base (from c.o)


        INCLUDE     "resources/misc.i"


        xdef   MiscName
MiscName    MISCNAME        ; macro from resources/misc.i

        xdef   _MiscResource
_MiscResource       dc.l   0; place to store misc.resource base

;parallel port hardware addresses (from amiga.lib)

        xref   _ciaaprb     ; the actual port address
        xref   _ciaaddrb    ; data direction register

        xref   _ciabpra     ; control lines are here
        xref   _ciabddra    ; data direction register
;from amiga.lib
        xref   _LVOOpenResource
        xref   _LVOAllocMiscResource
        xref   _LVOFreeMiscResource
```

```
_getport
;This routine simply allocates the parallel port in a system friendly
;way, and sets up the lines we want to use as input lines.
;


;save registers on the stack

      movem.l      a2-a6/d2-d7,-(sp); push regs


;open the misc.resource

      lea    MiscName,a1  ; put name of misc.resource in a1
      movea.l      _SysBase,a6; put SysBase in a6
      jsr    _LVOOpenResource(a6)
      move.l d0,_MiscResource; store address of misc.resource
      bne.s  grabit

;Oops, couldn't open misc.resource.  Sounds like big trouble to me.

      moveq  #20,d0       ; error code
      bra    done


;This is where we grab the hardware.  If some other task has allocated
;the parallel data port or the parallel control bits, this routine will
;return non-zero.

;This part grabs the port itself

grabit lea    Name,a1      ; The name of our app
      moveq  #MR_PARALLELPORT,d0; what we want
      movea.l      _MiscResource,a6; MiscResource Base is in A6
      jsr    _LVOAllocMiscResource(a6)
      move.l d0,d1
      beq.s  grab2

;well, somebody else must've got the port first.

      moveq  #30,d0       ; error code
      bra    done

;This part grabs the control bits (busy, pout, and sel.)
;We really don't need pout, but it comes free with PARALLELBITS,
;so we'll take it anyway.

grab2 lea    Name,a1      ; The name of our app
      moveq  #MR_PARALLELBITS,d0; what we want
      jsr    _LVOAllocMiscResource(a6)
      move.l d0,d1
      beq.s  setread

;well, somebody else must've got the bits first.

      moveq  #40,d2
      bra    freepar


;set up parallel port for reading

setread      move.b #0,_ciaaddrb; all lines read

      andi.b #$FF,_ciabddra; busy, pout, and sel. to read

;Well, we made it this far, so we've got exclusive access to
;the parallel port, and all the lines we want to use are
;set up.   From here we can just put back the regs and return to
;the caller.

      bra    done
```

```
;If something happened AFTER we got exclusive access to the parallel port,
;we'll need to let go of the port before we return the error.

freepar      moveq  #MR_PARALLELPORT,d0
      movea.l      _MiscResource,a6
      jsr    _LVOFreeMiscResource(a6)

      move.l d2,d0        ; put error code into d0


;Restore registers and return
;(error code is in d0)

done  movem.l (sp)+,a2-a6/d2-d7; pop regs
      rts


_freeport
;This routine just makes sure that we let go of the parallel port and
;control lines, so somebody else can use 'em, now that we're all done.
;
;PS - Don't call this one if you got an error from _getport, as some
;of the resources might not have been opened, etc.
;

;save registers on the stack

      movem.l      a2-a6/d2-d7,-(sp); push regs


;free control lines

      moveq  #MR_PARALLELBITS,d0
      movea.l      _MiscResource,a6
      jsr    _LVOFreeMiscResource(a6)


;free parallel port

      moveq  #MR_PARALLELPORT,d0
      movea.l      _MiscResource,a6
      jsr    _LVOFreeMiscResource(a6)


;Clean up, restore registers, and return

      movem.l (sp)+,a2-a6/d2-d7; pop regs
      rts


_read34
;All this routine does is copy the data from the ports to other addresses.
;
;In this case the destinations happens to be whatever C variables are
;pointed at by _portptr and _fireptr.
;

      movea.l      _portptr,a1; a1 now holds the destination
      move.b _ciaaprb,(a1); move byte from port to dest

      movea.l      _fireptr,a1; a1 now holds the destination
      move.b _ciabpra,(a1); move byte from port to dest

      rts

      end
```

Amiga Mail
Volume II
May/June 1992

Making CDTV Titles That Work on Amigas

by Ben Phister, CATS

In the coming months Commodore plans to begin delivery of the A570 drive. This CD-ROM peripheral for the Amiga 500 and Amiga 500 Plus should rapidly expand the installed base of CDTV-compatible players. With over 3 million Amigas sold worldwide, the potential for the A570 is enormous.

The A570 presents exciting opportunities for CDTV developers. It also presents a certain number of challenges. Great efforts have been made to ensure compatibility of titles for both platforms. However, there are numerous differences between the CDTV and the A570: keyboard plus mouse vs. remote control, TV sets in a living room vs. RGB monitors on a desktop, Kickstart 2.04 vs. Kickstart 1.3, etc.

With the availability of the optional keyboard for CDTV players, many users now attach a keyboard to their CDTV. Other users have attached CD-ROM peripherals to their Amiga 2000 or A3000 computers, and need keyboard support from titles.

To ensure strong sales of your title on all the systems capable of playing CDTV discs, your title should take advantage of the new features of the A570, yet remain easy-to-use on the CDTV player. This article discusses the differences between the two systems, and presents extensions to the *CDTV User Interface Guidelines* to allow titles to run on both platforms.

Physical Differences

The A570 does not include the following items found on the CDTV player:

- ❑ Infrared remote control.
- ❑ Audio CD control buttons (Play, FF, etc.).
- ❑ Fluorescent display.
- ❑ Personal RAM Card slot.
- ❑ RF or composite video output to TV or VCR.
- ❑ MIDI ports.

Of course, functional equivalents of most of these items are already a part of the Amiga 500; many others are available as options.

Although the Amiga lacks the standard CDTV resources above, the Amiga connected to the A570 provides resources not available on all CDTVs. Every title running on an A570 will have access to a floppy disk drive and a keyboard. CDTV titles should support saving to a floppy, along with saving to a personal RAM card.

Operating System Issues

Kickstart 1.3 vs. 2.04

The A570 connects to either an A500 running Kickstart 1.3 or 2.04, or an A500 Plus running Kickstart 2.04. It is imperative that your application work in both these environments. The A570 uses the Kickstart ROM routines found in the host computer. It contains its own ROMs which provide the CDTV-specific functions (*cdtv.device*, *playerprefs.library*, Audio control panel, CDTV Preferences, etc.)

What Am I Running On?

It is possible for applications to detect if they are running on an A570 or on a CDTV player. There is an A570-specific module called "A690ID". A title can test to see if this module is present using the *exec.library* function `FindResident()`. If the "A690ID" module is present, the title is running on an A570-equipped A500.

Memory

An Amiga with an A570 may have various amounts of RAM. In a CDTV the minimum memory configuration is 1 Mbyte of Chip RAM. Do not assume that all systems have the same memory configuration. Some users might try to run your application on an Amiga + A570 with only 512K of Chip RAM and no Fast RAM. Others may have a 512K Chip/512K Fast RAM system. To avoid crashing due to memory shortages, check the amount of memory available when your application begins execution. If not enough memory is available, you should display a message to that effect and stop operating rather than crashing the machine.

Others may have more than 1 Mbyte of RAM: either Fast or Chip memory. Your title should also work properly under these conditions.

Emulating the Remote Controller

One of the major differences between the A570 and the CDTV is the remote controller. The Amiga's keyboard, the mouse, and, optionally, the joystick replace the remote controller. Your title must take these differences into account, and your input handling routines may require modification. The following section will discuss these differences, and make additions to the *CDTV User Interface Guidelines*.

Here is a chart describing how Amiga keyboard and mouse actions should correspond to the CDTV player's remote controller buttons.

CDTV Remote Controller Emulation

Remote Controller	Amiga Keyboard	Amiga Mouse/Joystick
cursor buttons (directional arrows)	Cursor keys	Mouse movement
'A' button	Left Alt + Left Amiga, Left Amiga key alone, F9 function key, 'A' key on keyboard	Left button
'B' button	Right Alt + Right Amiga Right Amiga key alone, F10 function key 'B' key on keyboard	Right button
0-9 buttons	Keyboard 0-9 Keypad number keys	
ESCAPE	Esc and Help	
ENTER	Return or keypad Enter	
REW	F1	
PLAY	F2	
PAUSE	F3	
FF	F4	
STOP	F5	

Cursor Buttons

An application that runs on an A570 can emulate the four directional arrow buttons on the CDTV remote control using two methods: via mouse movements *and* via the four directional arrows next to the numeric keypad. Titles should support both methods of cursor control. Jumping highlights are easy to control using the arrow keys. If the title has a pointer, the user should be able to move it using the mouse, but, in case the user doesn't have a mouse, titles should also allow the user to move the pointer with the arrow keys.

The 'A' and 'B' Buttons

The A and B buttons on the remote correspond to the left and right buttons on the mouse. If the user prefers to use a keyboard for input, he can use any of the following:

The left Alt/left Amiga combination for the 'A' button and right Alt/right Amiga for the 'B' button.

The left Amiga key (raw keycode 66 hex) alone for 'A', and right Amiga key (raw keycode 67 hex) alone for 'B'.

The F9 function key (rawkey code 58 hex) for 'A' and F10 function key (rawkey code 59 hex) for 'B'.

the A (for left) and B (for right) keys on the keyboard.

By offering a keyboard equivalent to the 'A' and 'B' button, users with keyboards will not have to move their hands from the keyboard to the mouse in order to confirm a selection. It also avoids accidental movements of the mouse when pressing the rodent's left or right button.

The F9 and F10 function keys were selected because of their convenient location near the directional arrow keys on the keyboard.

Supporting the 'A' and 'B' keys on the keyboard is more complicated. The 'A' or 'B' key on the keyboard cannot be equivalents for the remote's A and B buttons during the input of a text string. In other situations, however, this mapping can be helpful to the neophyte user.

If a title uses the console device, it should support both uppercase and lowercase ('A' and 'a'). Furthermore, the raw keycodes for the 'A' and 'B' keys must always be translated via the active keymap to determine the actual ASCII value. The *rawkeyconvert.c* routines in the ROM Kernel manual give an example of this translation process.

If your title reads raw keycodes directly, beware of using raw keycodes for the 'A' and 'B' keys on the keyboard, as the raw keycodes may change if the user has a French keyboard. The French keyboard has reversed the position of the 'A' and the 'Q' keys. If you choose to have raw keycode support, you should accept both the 'A' (raw keycode 20 hex) and 'Q' (raw keycode 10 hex) for the 'A' button, and 'B' (raw keycode 35 hex) for the 'B' button.

0-9 Buttons

The 0-9 and ENTER buttons located in the center of the remote are replaced by the Amiga keyboard's numeric keypad. The keycodes of the remote represent those found on the keypad, the only difference being the reversed physical layout of the keys:

Remote	Numeric keypad
1 2 3	7 8 9
4 5 6 0	4 5 6
7 8 9	1 2 3
	0

Your code will probably not have to be modified to take this into account, unless it displays 9 icons in a grid pattern, and wants the user to type in a number corresponding to a certain position. If you support a jumping highlight (as suggested in the *CDTV User Interface Guidelines*), this should not be a problem.

Your code should also support the 10 numeric keys on top of the alphanumeric keyboard (just below the function keys) for input of numbers.

Escape Key

The ESCAPE key requires special attention with a keyboard. On the CDTV remote control, the ESCAPE key generates the ESCAPE rawkey code of 45 hex, the same code as on the A500's keyboard. The *CDTV User Interface Guidelines* recommend that titles use the ESCAPE button as a HELP key.

Help Key

The A500 keyboard has a Help key, rawkey code 5F. To run effectively on both the CDTV and A500 with an A570, applications should support both the Escape key and the Help key to obtain Help functions. If an application detects that the user has pressed either the Help key or the Escape key on the Amiga keyboard, the application should provide a help function to the user.

REW, PLAY, PAUSE, FF, STOP

These keys are mapped by the A570 ROM to the F1 through F5 keys on the Amiga keyboard. The Audio Control Panel uses these equivalents to play CD audio discs. If a title detects a keypress on F1 through F5, it should react as if the user had pressed the corresponding remote control button.

Certain titles ported from the Amiga to the CDTV may have already established alternate uses of the F1 through F5 keys on the keyboard. Any such application that retains its mapping of the F1 through F5 keys for its A570 version should clearly document these alternate key mappings in its manual.

Genlock

The Genlock button on the CDTV remote is for mode selection of an optional video genlock device that can be installed in the CDTV. This button has no equivalent on the A570. If you want to support this feature in an application running on an A570, you cannot be sure that the genlock used with the Amiga supports software control of mode switching. Again, mention this fact in the user manual.

CD/TV

This button on the CDTV remote controller determines whether the TV/monitor displays input from the CDTV or from the broadcast (or cable) TV source. This button has no equivalent on the A570. If the user has his Amiga connected to a TV set, his existing cabling handles this switch-over.

JOY/MOUSE

This button on the remote determines whether the remote's cursor buttons respond as a joystick or as a mouse. This is not necessary on the Amiga + A570: to enter joystick mode, the user simply plugs a joystick into the Amiga. To enter mouse mode, he uses his mouse.

Some CDTV applications only support one mode or the other. Titles should support both modes, if possible. If a title only supports joystick mode and the title cannot be reasonably controlled from the keyboard, you should indicate this on your packaging, as an Amiga + A570 user might not have a joystick.

Screen Design Issues

The typical CDTV user is sitting approximately 3 meters from his TV set. The typical A570 user will be sitting less than one meter from an RGB monitor. These differences require some foresight on the part of your application.

Fonts

For CDTV usage, a typical font should be no less than 18 points high. This permits the user to easily view text from a distance. An A570 user might prefer a smaller font in order to have more text on the screen, especially when using text-based applications. In this case, an application should offer a selection of at least 2 different-sized fonts to the user.

Three Levels of Support for the A570

Full support for the A570 implies numerous constraints for the CDTV developer. He has to worry about supporting a keyboard, working with different versions of the operating system, dealing with varying memory configurations and working around differences between Amiga and CDTV displays. The issues of starting, exiting, and multitasking titles remain.

There are several Amiga/CDTV configurations that may be booted into a CLI or Workbench environment: Amiga + A570, any Amiga computer equipped with a CD-ROM, and a CDTV player with a floppy or hard disk attached. In these cases, the user may want to launch a CDTV title from that environment. If he does so, he will expect the title to exit cleanly, release all memory, remove any special assigns that may have been executed, etc.

Users may even want to multitask a title. Most Amiga applications may be multitasked, and the user may well expect the same from a CDTV title. Reference titles would be significantly enhanced if they supported cutting-and-pasting directly into another application like a word processor. Those developing dictionary and encyclopedia titles may want to provide access methods for users or other applications to search on a key word. Some may even want to add an ARexx port.

These issues, while reasonable requests from a user, add significant new burdens to the developer. Many CDTV titles are designed to take over the machine. Many of these titles require all the memory available in the system (and even more, in some cases). They often make special assigns for fonts, data, etc. Many include no clean-up code.

Full compliance to AmigaDOS standards may be unreasonable for some titles. To address this problem, we have established three levels of compatibility for CDTV titles running on Amigas: *Minimal Support*, *Support For Workbench Startup and Exit*, and *Full Multitasking Support.*

Level One–Minimal Support

An application that meets Level One support requires only a few enhancements from a standard CDTV-only title.

1) Supports input from the keyboard and the mouse.

2) Runs under both Release 1.3 and Release 2 of the operating system (which is expected of all CDTV titles).

3) Includes a way for the user to start the title from a CLI and the Workbench. Such a title may reboot the system when it starts up. It may take over the entire system, refuse to multitask with other programs, have no cleanup code, and never exit.

Commodore has developed a standard icon and associated code, called *BootCD*, which developers may include on their CDTV disc. This icon will appear from the Workbench. If the user double-clicks it, a message will appear telling the user to remove any floppy disks from the drive, and warning that pressing the Continue gadget will reboot the system. If the user continues, *BootCD* will reset the system, and the system will then boot directly from your title on CD-ROM.

Level Two–Support For Workbench Startup and Exit

This level provides the user with a way to run CDTV applications without rebooting the system. A Level Two compliant title will also return cleanly to the Workbench or the CLI when it is finished. This stage requires that a title:

1) Supports input from the keyboard and the mouse.

2) Runs under both Release 1.3 and Release 2.0 of the operating system.

3) Launches from either the Workbench (via an icon) or the CLI.

4) Manages any resources it allocates.

5) Safely and Cleanly aborts in case any resource allocation fails.

6) Cleans up properly upon exit, closing all libraries and devices, and returning all memory.

7) Remembers the environment from which the user launched the title. If the user started the title from a CLI or Workbench, the title must be able to restore that environment cleanly. If the title was booted directly from the CD-ROM, before exiting, the title should take precautions against leaving the system in an unusable environment. For example, if a title makes all of the Workbench colors black using the *devs:system-configuration* file, the user should not be allowed to return to the CLI or Workbench, as the user will only see a black screen.

In level two compatible titles, temporary assigns are acceptable. The application may assign system directories (such as fonts:, sys:, etc.) to the CD during startup. If it does so the title must restore any of the assigns it reassigned.

A simple way to launch a title from a Workbench icon is using the *IconX* utility. This allows you to create a script that the system executes when the user clicks on the program icon. This script may contain any assigns that a title requires.

An easy way a title can tell if the user booted directly from the CD-ROM is checking for special command line arguments. Imagine your title is called "MyApp". When the user boots the system from the CD-ROM, the *startup-sequence* launches the application by using a special command line, for example: "MyApp ss". The "ss" tells the title that the system was booted from the CD-ROM. If the user starts the application from the CLI, he would simply type "MyApp". A title might even utilize another special command line option to tell if it was launched from an *IconX* script (i.e. "MyApp iconx"). Using this method a title can detect how the user started it and can choose an appropriate exit plan.

Level Three–Full Multitasking Support

A title conforming to Level Three support allows users to multitask freely (given sufficient memory). These titles must include support for points 1) through 7) above. Furthermore, Level Three titles may not re-assign any directory assigns that the system requires (i.e. fonts:, sys:, s:, etc.).

If a title in this category needs to access a particular font on the CD-ROM, it should not re-assign *fonts:* to *cd0:fonts*. Instead, it should refer to the full path name of the CD via the volume name of the disc (for example, *MyDisk:Data/MyFile*). You should not refer to the device name (cd0:) since this may vary, either with more than one CD-ROM peripheral attached, or a different device name for a third-party drive.

If your title does refer to a volume name, be careful when pre-mastering the title. Make sure you enter the correct volume name when running the ISO utility. If another company does the pre-mastering for you, make sure they use the correct volume name.

Conclusion

Full support for the A570 does require a certain amount of effort on behalf of the CDTV developer. Many of these points are also relevant for those CDTV owners that expand their unit to be an Amiga.

Remember, you are not obliged to support all these features. Many CDTV applications take over total control of the system, and are not designed to multitask. There is nothing wrong with doing this, even if you want your title to run on the A570. If you include Level One support, you can insist that your title be booted directly from the CD. It is better to have a title which runs perfectly on the CDTV than one which is awkward to use on both the CDTV and the A570.

Amiga Mail
Volume II
May/June 1992

From the Desk of Gail Wellington

This is not deja vu; it is not even an old issue of Amiga Mail. It's Gail Wellington, back in CATS and enjoying the privilege of sharing a few thoughts with you again.

Having been singularly focused on CDTV for over two years, it is wonderful to be able to spend more time on the broader picture of Amiga developments. I am enjoying using an Amiga 3000 running Release 2.0 of the operating system and using many of the wonderfully sophisticated programs you have been developing. I especially enjoy the more extensive uniformity of user interface which makes it easier to get up to speed on an application, the ARexx tricks to customize applications to make them more convenient for me, the performance improvements that are a result of more efficient coding of key functions and more extensive use of multitasking, and many of the other improvements. The best analogy I can use is to say that while I was nurturing "the Baby" from infancy to its first steps in the world, Amiga has gone from an adolescent that was still awkward in some areas, to the graceful, adult ballerina that we all dreamed of at the Lincoln Center launch. You are all to be congratulated!

The majority of my tasks are still CDTV related. The "Baby" still has some growing to do but now that its has gone mainstream, there are more people to help nurture it (as happens with any special project, CDTV has been turned over to the various divisions of Commodore for ongoing sales, marketing, support and development). Over the coming months you will see many new tools and techniques for CDTV developers. You may also see some of the relevant CDTV developments made available for Amiga multimedia applications.

Very shortly, you will see the first release of a formal CDTV Developers Reference Manual. Special mailouts will continue as needed in the form of materials to be added into the manual. If you do not receive these mailouts, but wish to, you must complete a questionnaire indicating your interest in CDTV. Contact CATS administration or your country support manager to receive this.

It has been an exciting two years and after a suitable breathing space, I look forward to another special project. Meanwhile, I am enjoying getting caught up on all the new developments. My favorite thing? Well, simple pleasures are sometimes the most enjoyable. I think the thing I like best is not having to wait for all the icons to be displayed before I can click on one. Now why didn't we do that sooner...

Best regards, and thanks for continuing to support Amiga (and CDTV) with your work.

-GW

Table of Contents

May/June 1992

I
Programming Practices and Standards
general purpose Amiga programming

Tag Items and Tag Lists . . . . . I - 1
Staff

2.0 Version Strings . . . . . I - 3
Carolyn Scheppner

The ASL Library. . . . . . I - 7
John Orr

2.0 Compatibility Problem Areas . . . . . I - 19
Carolyn Scheppner, Bryce Nesbitt, Peter Cherna, and Darren Greenwald

Release 2.0 Specific Functions and Commands . . . . . I - 35
Compiled by Carolyn Scheppner

Callback Hooks . . . . . I - 43
David Junod

Debugging with Enforcer and Mungwall . . . . . I - 47
Carolyn Scheppner

II
AmigaDOS
dos.library

Executing External Programs with Release 2.0 . . . . . II - 1
Carolyn Scheppner

AmigaDOS Packet Interface Specification . . . . . II - 5
John Toebes

Standard Command Line Parsing . . . . . II - 27
John Orr

Notification . . . . . II - 31
Ewout Walraven

Using the AmigaDOS Pattern Matching Functions . . . . . II - 37
Ewout Walraven

The 2.0 Dos.library Path Name Handling Functions . . . . . II - 43
Ewout Walraven

Directory Scanning . . . . . II - 49
Ewout Walraven

Writing a UserShell . . . . . II - 65
Randell Jesup

III
Exec
the multitasking executive, exec.library

Using SetFunction() in a Debugger . . . . . III - 1
Ewout Walraven

68040 Compatibility Warning . . . . . III - 11
Michael Sinz

IV
Intuition and Workbench
the Amiga user interface

Obtaining an Icon Image . . . . . IV - 1
David Junod

An Introduction to V36 Screens and Windows . . . . . IV - 3
Ewout Walraven

Opening Screens and Windows on Any Amiga . . . . . IV - 17
Ewout Walraven

Introduction to Boopsi . . . . . IV - 23
John Orr

Writing a Boopsi Image Class . . . . . IV - 41
David N. Junod

Boopsi's Transparent Base Classes . . . . . IV - 57
Peter Cherna

AppWindows, AppIcons, and AppMenuItems . . . . . IV - 59
Fred Mitchell and John Orr

Making Your Windows More Shiny and Manageable . . . . . IV - 69
Ewout Walraven

V
Graphics
animation, text, the Blitter, layers.library

Using the ECS Genlocking Features . . . . . V - 1
Ewout Walraven

Finding the Aspect Ratio . . . . . V - 11
Carolyn Scheppner

Amiga Font Scaling and Aspect Ratio . . . . . V - 15
John Orr

VI
Sound and Music
audio, speech, MIDI

VII
IFF
iffparse.library and other IFF topics

The DR2D FORM . . . . . VII - 1
Ross Cunniff and John Orr

ECS Display Modes and ILBM CAMG . . . . . VII - 17
Carolyn Scheppner

Developing New IFF FORMs and Chunks . . . . . VII - 21
Christian Ludwig

VIII
Unix and Networking
Unix SVR4, NFS, TCP/IP, Ethernet

Using the Amiga A3000UX as a NFS File Server . . . . . VIII - 1
David Miller

Keeping Time--Interval Timing in Amiga UNIX . . . . . VIII - 9
David Miller

Developing Network Applications for the Amiga . . . . . VIII - 17
Dale Larson

A Shared Socket Library Server and Client . . . . . VIII - 31
John Wiederhirn and John Orr

SANA-II Network Device Specification . . . . . VIII - 51
Brian Jackson, Dale Larson, Greg Miller, and Kenneth Dyke

IX
Timer, Serial, and Parallel
timer.device, serial port, parallel port

Four Can Play–Supporting Parallel Port Joysticks . . . . . **IX - 1**
Christian Ludwig

X
Output
printers, printing, Postscript

XI
Input
user input, Commodities Exchange

Introduction to Commodities Exchange . . . . . **XI - 1**
John Orr

XII
CDTV
Commodore Dynamic Total Vision

Guidelines for CDTV Applications . . . . . **XII - 1**
Staff

Making CDTV Titles That Work on Amigas . . . . . **XII - 9**
Ben Phister

XIII
Hardware
Cards, chips, connectors

Video Adapter Diagrams . . . . . **XIII - 1**
Adam Levin-Delson